Understanding a Novice Programmer's Progression of Reading and Summarizing Source Code

Andrew Morgan
Software Engineering Research and Empirical Studies Lab
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio 44555 USA
asmorgan@student.ysu.edu

Bonita Sharif
Software Engineering Research and Empirical Studies Lab
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio 44555 USA
bsharif@ysu.edu

Martha E. Crosby
Department of Information and Computer Sciences
University of Hawaii at Manoa
Honolulu, Hawaii 96822 USA
crosby@hawaii.edu

ABSTRACT

The paper presents observations over the course of three months on the patterns and strategies a novice programmer (DO 21) uses while reading source code. The programmer was asked to read and summarize a program after completing three sets of lessons from an online course. Results indicate that the method of reading source code gets harder as the novice attempts to comprehend more difficult concepts. The analysis is presented in the form of a case study.

Keywords

eye tracking, source code reading, program comprehension strategies, computer science education

1. INTRODUCTION

Most universities teach students to start writing code early in introductory programming classes, without teaching them to read the code for understanding first. The task of comprehending code and the process used to teach students this core skill is at least as important as the task of writing code. In order to understand the process of reading and understanding code, a team of researchers from Freie Universitat Berlin and the University of Eastern Finland organized the first workshop on analyzing the expert's gaze held in Finland in November 2013. This year, the focus of the Koli workshop is on analyzing gazes of novice programmers.

2. METHOD OVERVIEW

A brief description about the method and study is now given. All participants of the workshop were granted access to three eye tracking sessions (data, visualizations, and videos) of one novice. Each of the eye tracking sessions were held after the novice completed certain lessons (namely lesson 1, lesson 4, and lesson 6) from an online Introduction to Java Programming Udacity course[1]. The novice was asked to study the program for as long as they wished and then provide a summary. The novice is referred to as DO21 in the paper. An optional data set for another novice named EU10 was later provided, but was not mandatory for analysis. The novices were both female and didn't have much experience programming. Workshop participants were urged to describe the data in terms of stages of development and asked for general thoughts on how this type of data could be analyzed. The eye tracking sessions were conducted on May 5th, June 16th, and July 14th of 2014 respectively.

3. OUR PREDICTION

Before we had a chance to look at the data, we were asked about what we expect to find in the reading behavior and also what ideas we had on the progress that could be perceived. Our predictions follow. If the student has been progressing well through the course, we would expect to see the student getting better at comprehending the source code given and becoming more efficient within their analysis. The reading behavior should get more structured as the student progresses through the course. This more structured approach is further clarified as the individually unique technique the programmer uses to understand the presented source code. The approach, along the way, will become more structured as the programmer understands his or her own techniques to interpreting such code. This reading behavior should also then focus on the important parts of the code. What is important will vary based on what the task is. For example, a bug finding task would involve different reading behaviors compared to a task that just tells the subject to look over the code and give an overview. At the time of this prediction, we were not aware of the task (summarization) in this case. We also predicted that they might find the answer quicker after lesson 6 when compared to the one after lesson 1. Every programmer has a different workflow they follow, however given enough subjects, there should be some commonality that can be extracted. So how do we measure progress? We could record time to complete task as one example. Depending on how long these files are, we could possibility also segment them into intervals and compare them.

4. ANALYSIS

We now present our analysis of each of the three eye tracking sessions namely lesson 1, lesson 4, and lesson 6. These

[1]https://www.udacity.com/course/cs046

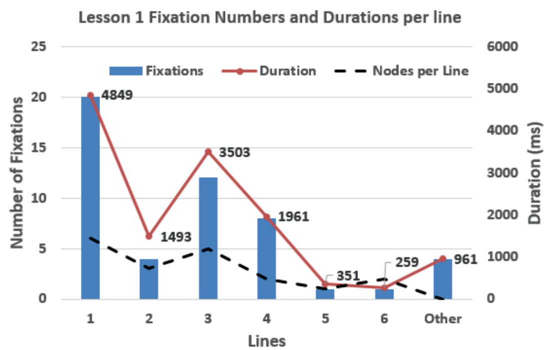

Figure 1: Fixations, Durations, and Nodes per line for Lesson 1 (pseudocode)

lessons are named after the last lesson the novice did online. For each of the lessons, we present a graph outlining the number of fixations, total duration, and nodes per source line and provide some discussion about them. Note that even though a line graph is used to show duration and nodes per line, there is no implicit connection between duration or nodes. The novice DO21 also provided an accurate summary (the main task considered for this workshop) after she read through all the code in each lesson.

4.1 *Lesson 1* Analysis

The first lesson's recording was taken after the novice had six days of online lessons. The six line program was written in pseudo code and contained a for loop with an embedded "if - else" statement. The novice seemed to read the code as though it were text. Lines 1 and 3 received the most fixations and also contained the most nodes. We refer to a node as an area of interest in the data files. For example, a node could be individual words and phrases contained in a statement. See Figure 1. We did not see any continual regressions, however, we noticed that she read the entire program twice. In this particular case, the time spent reading the lines correlated with the number of fixations on those lines (which is not always the case). This type of behavior is very similar to what we would expect of reading text in a natural language. We also noticed that there were 4 fixations totaling 961 ms that did not fall on any given line in the source code.

4.2 *Lesson 4* Analysis

After lesson 1, the novice learned about objects and classes. The eye-tracking recording for lesson 4 was done 42 days after the recording for lesson 1. Refer to Figure 2. This source code snippet contained a Scanner object in which input was saved and later used for showing the average on the screen. The program was 11 lines long (we excluded the last two lines with braces since no fixations were detected in that area). The last two lines were mainly brackets so it could be that the subject perceived with peripheral vision that the brackets were there or could have also taken for granted that the program was bug free with no need to check for braces. It is also possible that the student might have already learned that the braces were of little importance. Most of the fixations focused on lines 4, 5, 6, and 9. Line 4 created the Scanner object. Lines 5 and 6 read

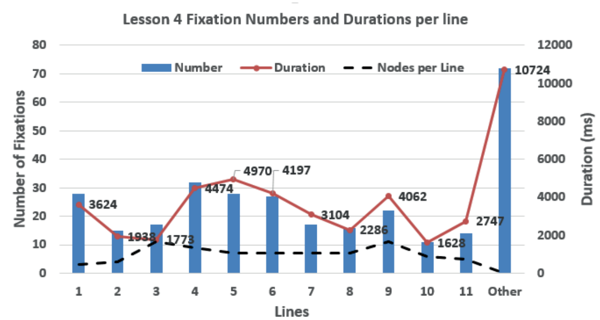

Figure 2: Fixations, Durations, and Nodes per line for Lesson 4 (CalculateAverage)

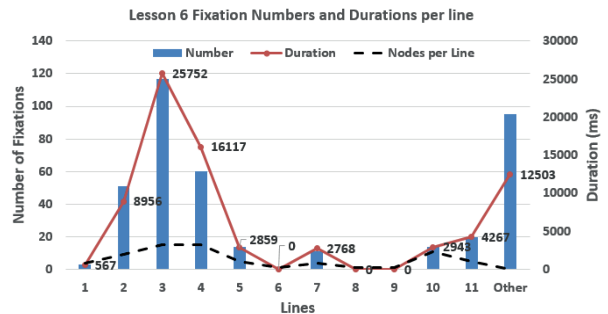

Figure 3: Fixations, Durations, and Nodes per line for Lesson 6 (PrintPattern)

in an integer and line 9 did the averaging of two numbers. If we compare the behavior of this subject with Lesson 1, it appears that this task was more difficult for her to solve. As DO21 tries to understand the code and build a mental model, she checks areas previously read. She reads through the program twice, i.e. we find two epics, from beginning to end. Between these two epics, we observed some sort of searching behavior. There were some regressions during the searching phase, where things were not looked at sequentially. The number of nodes, fixations and duration do not correlate in any particular way. In this lesson, there were 72 fixations (10,724 ms) that did not fall on any given line in the code.

4.3 *Lesson 6* Analysis

This lesson was recorded 29 days after Lesson 4. During this time, they covered decisions and loops. The source code contained a method called from within the main function. There was a nested for loop that printed stars in three rows with each row having one additional star than the previous. Refer to Figure 3. A lot of time is spent reading through the nested for loop in the method. Line 3 was the line most focused on, followed by line 4 and line 2 (method signature). DO21 spent nearly half a minute on line 3 throughout the session, which was the first for loop in the nested for construct. The total time spent in the method body was around 47 seconds (47,496 ms, 202 fixations) with about 9 seconds spent on the method signature (8,956 ms, 51 fixations), the third most looked at line in the program.

In this session, we counted about 7 epics (times DO21 went through the program from beginning to end). The

first time the novice read this program, the first focus was on lines 2 through 4 to understand what the method was doing. Later, the programmer proceeded to look at the main method. However, most of the gazes were focused on the method declaration's body. There was very little searching behavior and a lot more continual regressions between the lines in the nested for loop indicating a higher cognitive load because of higher task difficulty. In this lesson, there were 95 fixations totaling 12,503 ms that did not fall on any given line in the code.

4.4 Internal Testing for Further Analysis

A brief overview of the Java topics was introduced to a local novice programmer at Youngstown State University (Y10). In order to fully understand such data, the same tests were performed on this participant for each of the five source code snippets (combination of DO21's and EU10's). All regulations were similar to those for the workshop, and the programmer answered correctly to all summaries of the code. The only difference is that we conducted this small experiment all in one sitting.

When we compare Y10's eye gaze fixations to the two earlier subjects, we do see similar correlations. Y10's data had a tendency to experience Lesson 1 with a reading type behavior, while other lessons followed with a more problem solving type path with longer fixations and more focus on specific statements to understand meaning. Y10 answered all the summary questions correctly, however he took much longer time in terms of fixation duration for interpreting such code. In comparison, Y10 took up to two times the duration compared to DO21, and up to four times the duration as EU10.

5. STAGES OF DEVELOPMENT

Several studies describe the process of program comprehension but the evidence of *how* and *why* programmers perceive code is limited. Most studies explain *how* not *why* people read and comprehend programs. In the process of establishing a methodology for studying program comprehension, Weissman [25] found that initially students encountered problems with constructs of the programming language but eventually they were able to extract the programs meaning. By systematically investigating the effect on program comprehension of interactions between knowledge of the gist, features of the text and participant differences, it may be possible to determine when paradigm shifts (or stages as suggested by Flavell [7]) emerge.

Research suggests that stage shifts occur as novices become experts. Adelson [1] shows experts rely on abstract problem descriptions to understand code using semantics while novices are driven more by syntax and other categorization strategies. Davies [6], Gilmore and Green [8], Green and Navarro [10], Rist [18], Soloway and Ehrlich [21] and Bertholf and Scholtz, [3] argue that experienced programmers use programming plans during the comprehension process. Little is known about the progression of the processes involved as novices become experts. Evidence suggests that some people are more skilled than others, independent of the number of years programming [11]. However, the underlying reasons remain elusive.

Program comprehension has been described as 1) top-down by Brooks [4]; 2) bottom-up by Basili and Mills [2]; Shneiderman and Mayer) [20]; 3) knowledge based by Letovsky and Soloway [12], 4) as-needed by Littman et al. [13] and Soloway et al. [22]; 5) control-flow based by Green [9], Navarro-Prieto [14] and Pennington [15] and 6) integrated by von Mayrhauser [24]. Research by Clayton, et al. [5]. Shaft and Vessey [19] and von Mayrhauser and Vans [23] indicates the top-down approach is used to scan through source code. While bottom-up is used if people are unfamiliar with a particular application domain. While the integrated model of program comprehension is compelling, there is not clear evidence to support this model.

Application domain knowledge has been shown beneficial for program comprehension. People that are familiar with a domain tend to understand programs better than people that are not familiar with the domain [17]. Pennington [15], Petre et al. [16] and Navarro et al. [14] studied the mental representations used during program comprehension. Their studies present a model of how people build a mental image when trying to understand code. However, it is difficult to extract meaning from scan patterns alone. How do they relate to other studies that focus on models of program comprehension? Can scan patterns be classified in a meaningful way to clarify stages of comprehension? Comparing the scan patters of participants who understand the programs gist versus participants who do not may give insight into when paradigm shifts or stages occur.

6. SUGGESTIONS

One method of determining a novice's progression would be to show them source code that was similar to lesson 1 at lesson 4 and lesson 6. Similarly, it is necessary to show them source code similar to lesson 4 at lesson 6 in time. We can then see the learning that has occurred of the concepts learned at earlier sessions. Since this was not done in this study, we are not able to say for sure, but only guess as to what learning occurred. Another point is to design tasks that take advantage of the kind of mental structure they use to solve the problem. This can bring out the problem solving nature of the task that these programs analyzed did not have, even though they were semantically rich in syntax.

Regarding what points in time need to be examined more closely, the answer will really depend on what we are trying to determine. If our goal is to find stages in development then it is going to take a longer time to follow the person and have the person be their own control. This will help us determine when the novice actually exhibits expert like behavior (if ever) and this is the point at which we can say that the novice has started using "chunks" for example and behaves more like an expert. For example, we could determine if the novice has now started building hierarchical tree like representations to solve the task or if they still focus on a flattened out tree with no clue as to which path to take. This change of representation needs to come through with a good selection of tasks.

To fully understand the kind of data presented here requires multiple levels of analysis. The videos and fixation graphs by time do help. One thing that is also important is time spent at each fixation. The big circles are indicative of the task getting harder. Sometimes there could be a few fixations but a lot of time spent on them. Points of dis-connectivity in the fixation time line graphs also need to be examined (could imply cognitive load or thinking and reasoning).

We did not numerically analyze the additional EU10 dataset

since the programs used are syntactically and semantically a lot different at lesson 1 and lesson 4 compared to DO21. It would not be appropriate to compare them side by side. Even though the same program was used for lesson 6, EU10 did not summarize the code correctly, so we only provided a brief visual comparison.

7. DISCUSSION AND CONCLUSIONS

We predicted initially even before we saw the eye tracking sessions that the reading would become more structured. Reading pseudocode was more like a reading task, which in turn required two linear epics within, to understand. Whereas, the other two were also classified as reading but it was getting harder for DO21 to read the more complex constructs. DO21 did provide all correct summaries to the programs. The fact that the reading got more difficult can be seen in the data and videos of the sessions. It could be that the keywords used and structure of the program caused this to occur. Unfamiliarity with the methods can also cause this to happen.

We also noticed a lot of fixations that fell either on blank space on the screen or outside the screen (where the novice looked at something other than the computer screen). They may also have closed their eyes briefly to think about the task at hand. In the timeline graphs provided, these out-of-screen or out-of-line fixations can be seen as breaks in the line graphs. These breaks mainly occur during the searching that happens between the epics.

In visual comparison between Lesson 6 tests' between DO21 and EU10, we can again see the apparent time difference between the two. While DO21 appears to interpret presented code in a problem solving form, EU10 performs fewer epics on the code itself, along with spending less duration on the nested for-loop. The fixations for EU10 scattered the length of the code, compared to focusing such cognitive load on the main construct of the program. In return, the summarization of the code was answered incorrectly due to insufficient understanding within the for loop itself.

We were not able to identify clearly any stages in the videos or data. One might argue that each of the videos could be split into three phases where they first read the program and recognized it, followed by analyzing in detail, followed by a conclusive stage. However, this is not apparent while viewing the videos closely or looking at the numbers. It is too early in the learning process of a difficult skill to find such stages. It is unclear if they are chunking, for example. Stages are a profound shift in understanding and we didn't see this in the sessions presented.

Another possibility is to distinguish problem solving from reading. It might also appear that the novice is doing some form of problem solving in lesson 4 and lesson 6. Lesson 1 appears to be a reading task. However, in order for the problem solving theory to hold, they had to be working on a different type of task. Problem solving is a major research area in cognitive science. In programming, problem solving is a key skill. However, we do not believe the programs triggered any problem solving type of behavior. The task represented does not lend itself well to solving a problem where the participant requires to build and change their mental representation. In addition, eye movements alone cannot be used to show this. So we conclude that the novice DO21 is really in the process of reading and understanding the code in all lessons but due to the nature of the constructs used, things are getting harder to read. Do things get easier as time goes on? We didn't see it yet but it might in the longer future. This is when stages in behavior might become more apparent.

8. REFERENCES

[1] B. Adelson. *Structure and Strategy in the Semantically-Rich Domains*. PhD thesis, 1983.

[2] V. R. Basili and H. D. Mills. Understanding and documenting programs. *IEEE Trans. Software Eng*, 18:270–283, 1982.

[3] S. J. Bertholf, C. F. Program comprehension of literate programs by novice programmers. *Empirical Studies of Programmers: Fifth Workshop*, page 222, 1993.

[4] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[5] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. *Working Conference on Reverse Engineering*, 1998.

[6] S. P. Davies. The nature and development of programming plans. *International Journal of Man-Machine Studies*, 32:461–481, 1990.

[7] J. H. Flavell. Stage-related properties of cognitive development. *Cognitive Psychology*, 2:421–453.

[8] D. J. Gilmore and T. R. G. Green. Programming plans and programming expertise, the quarterly. *Journal of Experimental Psychology*, 40A(3):423–442, 1988.

[9] T. R. G. Green. Cognitive approaches to software comprehension: results, gaps and limitations. *Extended abstract of talk at workshop on Experimental Psychology in Software Comprehension Studies 97*, 1997.

[10] T. R. G. Green and R. Navarro. Programming plans, imagery, and visual programming. *In Nordby, K., Helmersen, P. H., Gilmore, D. J., Arnesen, S. (Eds.) INTERACT-95*, pages 139–144, 1995.

[11] A. J. Ko and B. Uttl. Individual differences in program comprehension strategies in unfamiliar programming systems. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, 2003.

[12] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 41–49, 1986.

[13] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, Dec. 1987.

[14] R. Navarro-Prieto. Mental representation and imagery in program comprehension. *Psychology of Programming Interest Group, 11th Annual Workshop.*, 1999.

[15] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, pages 295–341, 1987.

[16] B. A. F. Petre, Marian. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, pages 7– 30, 1999.

[17] V. Ramalingam and S. Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. *7th Workshop on Empirical Studies of Programmers*, 1997.

[18] R. S. Rist. Plans in programming: Definition, demonstration and development. *In E. Soloway and S. Iyengar (Eds.), Empirical Studies of Programmers*, 1986.

[19] T. M. Shaft and I. Vessey. The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research*, 6:286–299, 1995.

[20] B. Shneiderman and R. Mayer. Syntactic semantic interactions in programmer behavior: A model and experimental results. *Intl. J. Comp. and Info. Sciences*, 18:219–238, 1979.

[21] E. Soloway and K. Ehrlich. Plans in programming: Definition, demonstration and development. *Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering*, pages 595–609, 1984.

[22] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31:1259–1267, 1988.

[23] A. von Mayrhauser and A. Vans. Comprehension processes during large scale maintenance. *16th International Conference on Software Engineering*, 1994.

[24] A. von Mayrhauser and A. Vans. Program understanding: Models and experiments. *Advances in Computers, M. C. Yovits and M. V. Zelkowitz, Eds. Academic Press Limited*, 40, 1995.

[25] L. Weissman. *A methodology for studying the psychological complexity of computer programs.* PhD thesis, 1974.